Modélisation et Détection Formelles de Vulnérabilités Logicielles par le Test Passif†

Amel Mammar[1] et Ana Cavalli[1] et Edgardo Montes de Oca[2]
et Shanai Ardi[3] et David Byers[3] et Nahid Shahmehri[3]

[1]*Télécom & Management SudParis, 9 rue Charles Fourier, 91011 Evry Cedex, France*
E-mail: {`amel.mammar,ana.cavalli`*}*`@it-sudparis.eu`
[2]*Montimage, 39 rue Bobillot Paris 75013, France*
E-mail: `edgardo.montesdeoca@montimage.com`
[3]*Department of computer and information science*
Linköpings universitet, SE-58183 Linköping, Sweden
E-mail: {`shaar,davby,nahsh`*}*`@ida.liu.se`

L'utilisation de modélisations formelles est devenue une partie intégrante du processus de développement de logiciels sûrs. En effet, une bonne modélisation du système à développer permet d'améliorer la qualité des logiciels en détectant, par exemple, certaines vulnérabilités avant même leurs déploiements. Dans cette optique, ce papier propose une nouvelle méthode de modélisation de vulnérabilités ainsi qu'un langage formel pour l'expression précise sans ambiguïté des causes et événements pouvant les produire. La définition d'un tel langage formel permet également la détection automatique des vulnérabilités par des outils de test. Plus précisément, nous illustrons l'utilisation de l'outil de test passif *TestInv*, développé au sein de notre équipe, pour la détection automatique de vulnérabilités exprimées dans le langage formel ainsi défini. Notre approche a l'avantage de produire un nombre beaucoup plus réduit de faux positifs tout en maintenant à jour la base de connaissances de l'outil *TestInv*. L'approche proposée est illustrée à travers l'exemple de vulnérabilité CVE-2005-3192 représentant un "buffer overflow" dans un programme C.

Mots-clés: Vulnérabilité, modélisation, spécification formelle, détection automatique.

---

1 Introduction

L'impact important voire catastrophique de vulnérabilités -failles de sécurité- dans un système a amené les utilisateurs à multiplier leurs efforts pour la définition de méthodes et le développement d'outils pour la détection et l'élimination de ce type d'erreurs de programmation dès les phases préliminaires de conception de tout logiciel. Actuellement, il existe un nombre important de techniques et d'outils qui contribuent à l'amélioration de la qualité du logiciel. Comme exemples de ces techniques et outils relativement utilisés dans l'industrie, on peut citer les méthodes formelles, les techniques de vérification et de validation et également les analyseurs statiques et dynamiques de codes [S. 04, KMC06]. Nos travaux de recherche portent plus particulièrement sur les techniques de modélisation et de détection formelles de vulnérabilités. Dans ce domaine, les approches existantes sont peu nombreuses et ne se basent pas toujours sur une modélisation formelle précise des vulnérabilités qu'elles traitent [Cov08, For08, Klo08]. De plus, les outils de détection sous-jacents produisent un nombre conséquent de faux positifs et de faux négatifs. Notons également qu'il est assez difficile pour un utilisateur de savoir quelles vulnérabilités sont détectées par chaque outil vu que ces derniers sont très peu documentés.

Pour répondre au besoin de modélisation de vulnérabilités logicielles, une méthode graphique a été développée à l'Université de Linköping [ABS06, BASD06, BS07]. Cette méthode consiste à identifier les

†The research leading to these results has received funding from the European CommunityŠs Seventh Framework Program (FP 7/2007-203) under the grant agreement number 215995 (http://www.shields-project.eu/)

différents événements et conditions (causes) qui peuvent potentiellement produire une vulnérabilité donnée. Ces causes sont ensuite représentées sous forme de graphe orienté appelé *graphe de causes de la vulnérabilité* (VCG pour *vulnerability cause graph*). Les VCGs ont l'avantage de donner une vue à la fois intuitive et synthétique des différents scénarios qui produisent la vulnérabilité. Ils facilitent également la communication entre les différents acteurs participant au développement du logiciel. En dépit de ces avantages indéniables, l'aspect semi-formel des VCG ne permet pas l'utilisation d'outil pour la détection automatique de présence de vulnérabilité dans un code. Pour pallier à cet inconvénient, le présent article propose un langage formel, appelé *Condition de Détection de Vulnérabilité* (VDC pour *Vulnerability Detection Condition*), qui permet d'associer une description formelle précise et non ambigüe à chaque VCG en traduisant ses différentes causes en prédicats logiques sur lesquels il devient possible de raisonner. L'obtention d'une telle description formelle facilite le développement d'outils pour la détection automatique de vulnérabilités. Le formalisme de VDC est inspiré des travaux antérieurs de Télécom & Management SudParis et Montimage sur le test passif des protocoles [BCNZ05, CMML08]. La technique de test passif s'est avérée très efficace pour la détection de fautes; une application de cette approche pour le protocole WAP est présentée dans [BCNZ05]. Dans cet article, nous montrons l'utilisation de l'outil *TestInv*, supportant cette technique, pour la détection de vulnérabilités.

En résumé, les contributions principales de ce travail sont les suivantes :

- Un formalisme graphique VCG et un langage formel VDC pour la modélisation des vulnérabilités (Sections 2 et 3).
- Une approche de génération de l'expression formelle VDC d'une vulnérabilité à partir de sa modélisation graphique VCG (Section 3).
- Une méthode d'utilisation des VDCs pour la détection automatique de vulnérabilités (Section 4).
- L'outil de test passif *TestInv* et son application à un programme C contenant la vulnérabilité " buffer overflow ". Cette vulnérabilité a été préalablement modélisée par un VCG pour lequel un VDC a été généré (Section 4).

2 Modélisation des vulnérabilités

La modélisation des vulnérabilités [BASD06] désigne la méthode structurée de description des causes des vulnérabilités connues et récurrentes. Cette méthode consiste à analyser la vulnérabilité en question afin de déterminer les conditions et les événements pouvant produire la vulnérabilité considérée. Ces conditions et événements sont ensuite structurés sous forme de graphe appelé *graphe de causes de la vulnérabilité* (VCG pour *vulnerability cause graph*). La méthode de modélisation de vulnérabilités est similaire à celle d'analyse des causes racines dans le processus d'identification des causes à différents niveaux comme l'implémentation, la conception, les besoins, etc. [ABS06].

Un VCG est un graphe acyclique orienté composé d'un nœud *exit* représentant la vulnérabilité à modéliser et d'un certain nombre de nœuds causes dénotant des conditions ou événements pouvant produire la vulnérabilité considérée. Il existe trois types de nœuds causes: simples représentant des conditions ou événements simples, composites représentant une combinaison de conditions ou d'événements modélisés par un VCG, conjonctions représentant l'effet conjoint de deux ou plusieurs nœuds causes. Les arcs d'un VCG modélisent la succession des causes menant à la vulnérabilité.

Le VCG modélisant la vulnérabilité CVE-2005-3192 (buffer overflow dans le viewver xpdf) est représenté par la Figure 1. Les nœuds "Utilisation de buffers non adaptatifs" et "Utilisation non sûre de malloc" sont des exemples de nœud simple et composé respectivement.

3 Description formelle des causes

Les VCG sont non seulement utilisés pour la description des scénarios à risque menant à l'apparition de vulnérabilités, ils peuvent être exploités pour détecter les causes responsables de ces vulnérabilités. Afin qu'un

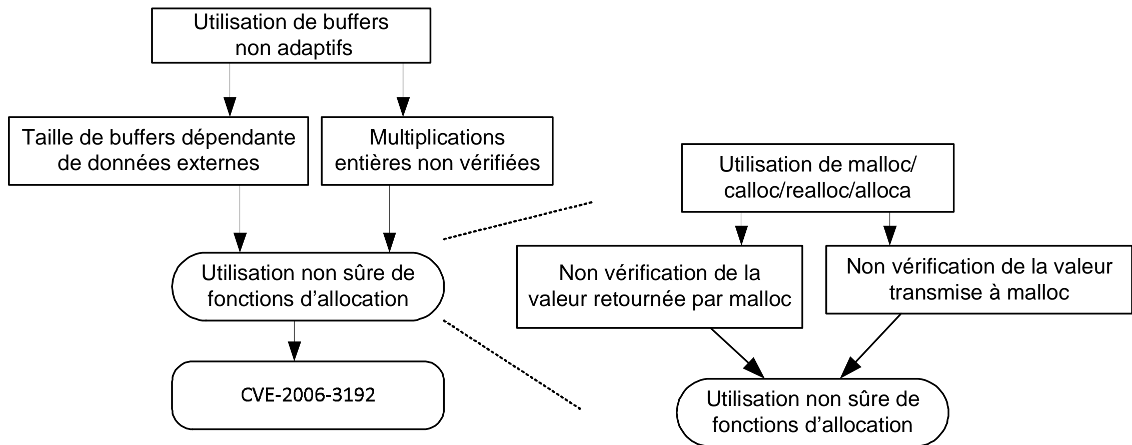

Figure 1: Graphe de causes de la vulnérabilité CVE-2005-3192 avec nœuds simples et composites

outil de test puisse détecter automatiquement la présence de ces causes, celles-ci doivent être formellement définies. Notons qu'une cause représente n'importe quel fait incluant des conditions ou des événements qui peuvent être interprétés ou pas par un outil automatique (ex. utilisation d'une fonction particulière, qualité des documents, etc.). Par conséquent, il n'est pas toujours possible d'associer une sémantique formelle aux différentes causes d'une vulnérabilité donnée. Par conséquent, les outils ne seront capables de détecter qu'un sous ensemble des causes responsables des vulnérabilités. Dans cette section, nous décrivons un formalisme permettant de décrire sans ambiguïté les causes apparaissant dans un VCG représentant une vulnérabilité.

3.1 Condition de Détection de Vulnérabilité

Un langage formel, appelé *invariants*, a été déjà défini par Bayse et. al [BCNZ05]. Basé sur les expressions régulières, ce langage permet d'exprimer des propriétés comportementales sur des systèmes communicants du domaine des télécommunications et des protocoles. Il s'agit, par exemple, de spécifier que l'occurrence d'un événement particulier e_1 est toujours précédée par l'apparition d'un autre événement e_2. Dans cet article, nous proposons d'étendre ce langage d'invariants afin de pouvoir exprimer formellement les causes d'un VCG. En effet, on devrait être capable d'exprimer des propriétés telle que:"*un espace mémoire alloué dynamiquement ne doit pas être utilisé (lu ou écrit) sans avoir vérifié préalablement le succès de l'opération d'allocation"*. Pour cela, nous proposons dans ce papier une adaptation du langage d'invariants que nous désignons par *Condition de détection de Vulnérabilité* (VDC pour *Vulnerability Detection Condition*). L'idée clé du concept de VDC est de détecter l'utilisation d'une action (fonction) dangereuse sous certaines conditions particulières. Par exemple, il est dangereux d'utiliser un espace mémoire non alloué. La définition formelle du concept de VDC est comme suit.

Définition 1 *(Condition de Détection de Vulnérabilité). Soient Act un ensemble de noms d'actions,Var un ensemble de variables, et P un ensemble de prédicats sur $(Var \cup Act)$. Une condition de détection de vulnérabilité Vdc est définie formellement par la grammaire BNF suivante(les longs crochets désignent des éléments optionnels):*

$$Vdc ::= a/P(Var,Act) | a[/P(Var,Act)];P'(Var,Act)$$

avec a représentant une action, $P(Var,Act)$ et $P'(Var,Act)$ désignent des prédicats sur les variables Var et les actions Act. Le VDC $a/P(Var,Act)$ signifie que l'action a est exécutée sous des conditions spécifiques représentées par le prédicat $P(Var,Act)$. Similairement, le VDC $a[/P(Var,Act)];P'(Var,Act))$ représente l'exécution de a, sous les conditions $P(Var,Act)$, suivie d'une instruction dont l'exécution satisfait $P'(Var;Act)$. Bien évidement, si l'action a n'est suivie d'aucune autre action, le prédicat $P'(Var,Act)$ est considéré comme vérifié.

Des conditions de détection de vulnérabilités plus complexes peuvent être définies inductivement en utilisant les différents connecteurs logiques.

Définition 2 *(Conditions de Détection des Vulnérabilités: cas général). Si Vdc_1 et Vdc_2 sont deux conditions de détection des vulnérabilités, alors $(Vdc_1 \vee Vdc_2)$ et $(Vdc_1 \wedge Vdc_2)$ sont également des conditions de détection des vulnérabilités.*

3.2 Quelques exemples de VDCs

La condition de détection de vulnérabilité Vdc_1 suivante peut être utilisée pour détecter l'accès à une zone mémoire non allouée ou libérée. Le prédicat $Assign(x,y)$ dénote l'assignation d'une valeur y à une variable mémoire x, $IsNotAllocated$ vérifie si la mémoire x est non allouée:

$$Vdc_1 = Assign(x,y)/IsNot_Allocated(x)$$

Dans les langages de programmation comme C/C++, certaines fonctions peuvent entraîner des vulnérabilités si ces dernières sont appliquées hors limites de ses arguments. L'utilisation de variables entachées (tainted en anglais) comme argument d'une fonction d'allocation mémoire est un exemple bien connu d'une telle vulnérabilité exprimée par la condition de détection de vulnérabilité Vdc_2 ci-dessous. Une variable est dite entachée si sa valeur est issue d'une source non sûre. Une telle valeur peut être obtenue à partir d'une lecture dans un fichier, d'une entrée de l'utilisateur, etc.

$$Vdc_2 = memoryAllocation(S)/tainted(S)$$

Une bonne pratique de la programmation consiste à vérifier la valeur retournée par une fonction d'allocation mémoire même si cette dernière n'est pas utilisée dans le programme. La condition de détection de vulnérabilité Vdc_3 permet de détecter l'absence d'une telle instruction de vérification:

$$Vdc_3 = (u := memoryAllocation(S)); notChecked(u, null)$$

3.3 Génération formelle de VDC à partir de VCG

En utilisant le concept de conditions de détection de vulnérabilité (VDC) défini dans la section précédente, il est possible d'associer une description formelle pour un VCG représentant une vulnérabilité. La traduction d'un VCG en un VDC est réalisée par les étapes suivantes:

1. Identification des causes quantitatives. Les causes d'un VCG sont classifiées en deux catégories:

   Causes quantitatives. Elles peuvent être détectées et vérifiées suivant un processus formel sans intervention humaine. L'utilisation d'une fonction particulière d'un langage, comme l'allocation mémoire, est un exemple de cette catégorie.

   Causes qualitatives. Elles ne peuvent pas être évaluées ou vérifiées sans intervention humaine. L'évaluation de ce type de causes est complètement subjective et peut différer d'une personne à une autre. L'appréciation de la clarté de rédaction d'un document est un exemple de cette catégorie car elle dépend du niveau d'expertise du relecteur.

   Comme seules les causes quantitatives peuvent être détectées automatiquement, la traduction d'un VCG en VDC concerne uniquement ce type de causes.

2. Identification des scénarios. Dans le VCG, chaque chemin ayant pour cible le nœud *exit* représente un scénario potentiellement dangereux.

3. Identification de l'action maîtresse et des autres actions de chaque scénario. Les causes quantitatives peuvent être classées en actions et conditions.

   Une action désigne un point particulier dans le programme où une tâche ou une instruction modifiant la valeur d'un objet donné est exécutée. Comme exemples d'actions, nous pouvons citer l'affectation de variables, la copie d'une zone mémoire, l'ouverture d'un fichier, etc.

Une condition désigne un état particulier du programme défini par la valeur et le statut de chaque variable. Pour un buffer par exemple, nous pouvons déterminer s'il est alloué ou pas.

Chaque scénario doit contenir une action maître *Act_Master* qui produit la vulnérabilité. Tous les autres nœuds de ce scénario représentent des conditions notées $\{C_1, \ldots, C_n\}$ sous lesquelles l'action *Act_Master* est exécutée.

Parmi ces conditions, une condition particulière C_k, appelée *condition omise*, peut exister. Cette condition doit être satisfaite par l'éventuelle action suivant l'action *Act_Master*.

4. Définition formelle des prédicats: l'utilisateur doit exprimer chaque condition par un prédicat formel.

5. En se basant sur les définitions 1 et 2, expression de la condition de détection de vulnérabilité associée à chaque scénario. Soient $\{P_1, \ldots, P_k, \ldots, P_n\}$ les prédicats modélisant les conditions $\{C_1, \ldots, C_k, \ldots, C_n\}$. La condition de détection de vulnérabilité exprimant formellement ce scénario dangereux est définie par:

$$Act_Master/(P_1 \wedge \ldots \wedge P_{k-1} \wedge P_{k+1} \ldots \wedge P_n); P_k$$

6. Définition du VDC global représentant le VCG comme la disjonction des VDCs associés aux différents scénarios (Vdc_i désigne le VDC associé à chaque scénario i):

$$Vdc_1 \vee \ldots \vee Vdc_n$$

Considérons par exemple la vulnérabilité CVE-2005-3192 de débordement mémoire associée au viewer PDF xpdf. Cette vulnérabilité est due à la copie d'une donnée utilisateur dans une zone mémoire (buffer) allouée dynamiquement sans avoir vérifié au préalable la taille de la donnée par rapport à celle de la zone. En effet, un utilisateur malveillant pourrait exploiter cette faille en exécutant un code particulier dans l'espace de l'application. Ainsi, l'attaquant peut avoir accès à la machine et aux données critiques qu'elle peut contenir. Le code le plus vulnérable a été trouvé dans la fonction texttttStreamPredictor:: StreamPredictor utilisée dans le fichier texttttStream.c.

Comme indiqué par le VCG associé à cette vulnérabilité (Voir Figure 2), la donnée entière fournie par l'utilisateur n'est pas vérifiée quand cette dernière est utilisée par la fonction *StreamPredictor* pour le calcul de la taille d'un buffer. Les différentes causes du VCG sont numérotées afin de pouvoir les référencer plus facilement dans la suite du papier.

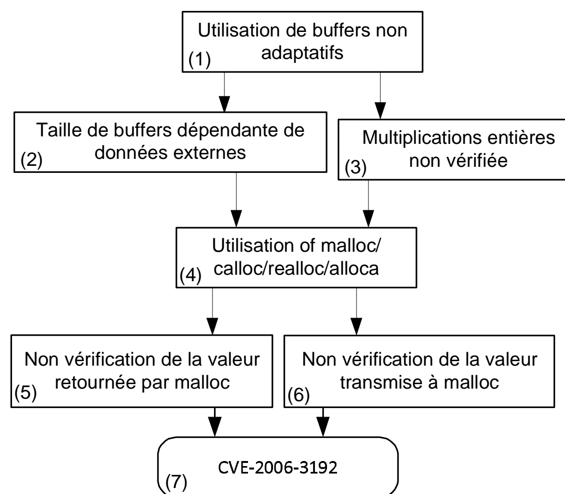

Figure 2: Graphe de causes de la vulnérabilité du viewer PDF xpdf

D'après la Figure 2, quatre scénarios sont possibles pour produire la vulnérabilité considérée:

$$(1,2,4,5,7),(1,2,4,6,7)$$
$$(1,3,4,5,7),(1,3,4,6,7)$$

Nous devons donc définir le VDC associé à chacun de ces scénarios. La méthode de définition des Vdcs étant similaire pour tous ces scénarios, seul le calcul du Vdc du premier scénario est présenté dans ce papier. Pour le scénario $(1,2,4,5,7)$, l'action maîtresse qui peut conduire à une vulnérabilité est l'utilisation d'une fonction d'allocation mémoire (nœud 4) sous les conditions suivantes:

L'usage de buffers non adaptatifs : le programme utilise des buffers dont les tailles sont fixées à l'exécution ou la compilation. Ce type de buffers ne peut contenir qu'une quantité limitée de données. Par conséquent, toute tentative d'écriture au delà de cette capacité conduirait à un débordement mémoire. Contrairement à ce type de buffers, les buffers adaptatifs ont la possibilité d'adapter leur taille pour contenir la donnée assignée. Pour chaque buffer non adaptatif, le prédicat suivant est vérifié:

$$nonAdaptiveBuffer(B)$$

La taille du buffer est dépendante de données externes: la taille d'un buffer alloué dynamiquement est calculée, entre autres, à partir de données utilisateur pour permettre à ce dernier de manipuler sa taille. Si une taille importante de ce buffer peut engendrer un déni de service, une taille trop petite peut conduire également à un débordement mémoire. Pour chaque buffer B dont la valeur est obtenue d'une source non sécurisée, le prédicat suivant est vérifié:

$$tainted(B)$$

Utilisation de malloc/calloc/realloc: le code considéré utilise des fonctions de gestion de mémoire similaires à celles du langage C telles que `malloc`, `calloc` ou `realloc` pour l'allocation de mémoire. Pour la fonction d'allocation mémoire f, le prédicat suivant est vérifié:

$$memoryAllocation(f)$$

La valeur retournée par la fonction *malloc* n'est pas vérifiée: le programme ne contient pas de mécanisme pour traiter de manière sûre le défaut de mémoire (i.e. traiter les valeurs nulles retournées par la fonction `malloc`). Exécuter des programmes sujets à des débordements mémoires non contrôlés peut générer des comportements imprévisibles qui peuvent être exploités par des utilisateurs malveillants. Cette cause est détectée quand la valeur u retournée par une fonction d'allocation n'est pas vérifiée pour savoir si elle est nulle ou pas. Par conséquent, pour chaque valeur u retournée par une fonction d'allocation mémoire, le formule suivante est définie:

$$notChecked(u, null)$$

Enfin, le VDC modélisant le scénario $(1,2,4,5,7)$ est défini par:

$$u := f(B) / \begin{pmatrix} memoryAllocation(f) \\ \wedge \\ nonAdaptiveBuffer(B) \\ \wedge \\ tainted(B) \end{pmatrix} ; notChecked(u, null)$$

Le VDC ci-dessus exprime une potentielle vulnérabilité quand:

1. une fonction donnée d'allocation mémoire f est utilisée avec un buffer non adaptatif B dont la valeur est produite d'une source non sûre,
2. la valeur retournée par la fonction f n'est pas vérifiée pour s'assurer qu'elle n'est pas nulle.

4 Détection des vulnérabilités par les VDCs

Dans cette section, nous illustrons notre approche de détection de vulnérabilité basée sur le test passif et la notion de VDC définie précédemment. Nous présentons également une application de cette approche pour montrer comment il est possible de détecter l'occurrence de la vulnérabilité CVE-2005-3192 par l'outil de test *TestInv* développé par Montimage en collaboration avec l'équipe de recherche de Télécom & Management SudParis [CMML08]. *TestInv* est aujourd'hui capable d'analyser des traces exécutables produites durant l'exécution d'un programme et de faire le lien entre les traces et le code source du programme étudié.

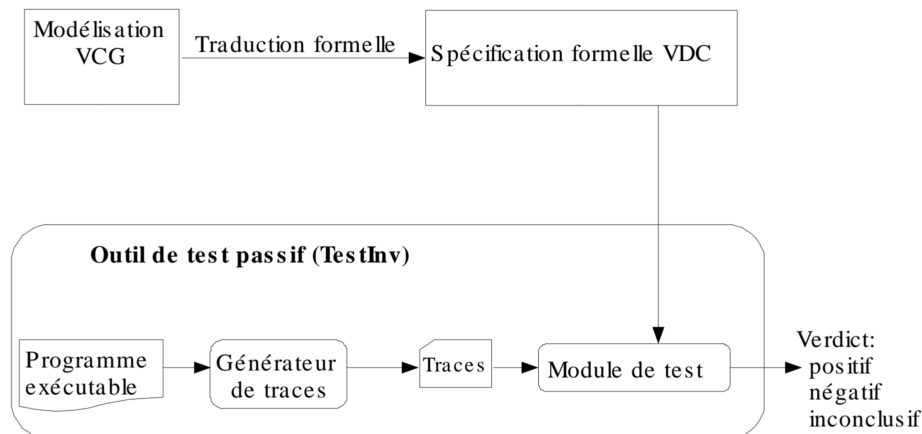

Figure 3: Test passif pour la détection de vulnérabilités

4.1 Test passif pour la détection de vulnérabilités

Le test passif a pour objectif de détecter des erreurs dans un système en observant uniquement ses entrées/sorties (ou d'autres caractéristiques observables) sans perturber son fonctionnement normal. Cette technique de test consiste à collecter des traces d'exécution du système (une partie des instructions exécutées) et détecter les éventuelles erreurs ou déviations de ces traces vis-à-vis du modèle formel du système. Le modèle formel peut être une spécification du système [ACC$^+$04, LNS$^+$97, MA01], des propriétés formelles que ce dernier doit satisfaire, ou des formules de VDCs comme c'est le cas dans le présent papier. La Figure 3 montre le processus de test passif pour la détection de vulnérabilités. Comme outil de test passif, *TestInv* accepte comme entrée des formules VDCs afin de détecter la présence des vulnérabilités considérées dans des traces d'exécution d'un système. La détection de vulnérabilités procède comme suit:

1. *Modélisation de la vulnérabilité:* la vulnérabilité à détecter est représentée par un VCG.

2. *Traduction du VCG en VDC:* cette étape est réalisée par un expert ayant la connaissance nécessaire du langage cible mais également du code assembleur généré par le compilateur utilisé. Cet expert doit pouvoir identifier les mots clés et les concepts qu'une trace d'exécution peut contenir.

3. *Extraction des traces d'exécution:* en utilisant l'outil *TestInv*, des traces d'exécution contenant toutes les informations nécessaires au test des formules VDCs sont analysées. L'outil permet de déduire différentes informations sur les variables du programme: le caractère sûr ou pas de la source d'une donnée, c'est-à-dire le caractère non teinté ou teinté de la donnée, le fait qu'une variable soit initialisée ou pas, contrôlée ou pas, etc. Comme indiqué dans la Figure 3, l'outil prend en charge l'exécution du programme, produit des traces pour chaque instruction (desassemblage) et le module de test analyse ces traces, garde les informations sur les variables et utilise les VDCs pour déterminer s'il y a une vulnérabilité.

4. *Vérification de la présence de vulnérabilité:* finalement, la détection de la présence éventuelle de la vulnérabilité est opérée par l'outil *TestInv* qui produit un diagnostic avec le numéro de ligne dans le

code (C/C++). Pour pouvoir déterminer la présence d'une vulnerabilité, il est nécessaire de compiler le programme avec des informations pour le deboggage. Si l'information n'est pas disponible, alors l'outil indiquera seulement l'addresse de l'instruction dans l'exécutable. En fonction de la présence ou pas de la vulnérabilité, le résultat est soit positif, soit négatif. Il peut être également inconclusif si l'outil ne connaît pas toutes les informations nécessaires à l'évaluation des VDCs.

Il est important de noter que l'étape 1 (resp. étape 2), bien que manuelle, n'est réalisée qu'une seule fois à chaque nouvelle découverte de vulnérabilité (resp. cause). De plus, les étapes 3 et 4 peuvent être semi-automatisées si des informations nécessaires à l'évaluation des VDCs, telle que l'adaptativité ou non d'un buffer, ne peuvent être obtenues que par l'utilisateur.

4.2 Étude de cas

Dans cette section, nous illustrons l'application de notre approche sur un petit programme C contenant une vulnérabilité de débordement mémoire similaire à celle du CVE-2005-3192. Pour détecter la vulnérabilité, notre outil doit être capable de déterminer:

- si une fonction d'allocation mémoire est utilisée grâce à une liste prédéfinie de fonctions pour chaque langage de programmation,
- les différentes variables destinées à recevoir les valeurs de retour de ces fonctions,
- si une variable donnée est teintée ou pas en utilisant le générateur de traces,
- si la taille d'une variable donnée a été vérifiée ou pas[‡].

C code	TestInv
size_t bytes_read;	
char tmp[7];	
bytes_read = read(fd, tmp, 6);	*tmp* et *bytes_read* sont teintés
tmp[bytes_read]='\0';	utilisation d'une variable teinté comme position pour changer un buffer sans la vérifier avant
int i=atoi(tmp);	*i* est teintée
char *buff;	
buff=malloc(i);	malloc dépend de la valeur de *i* sans la vérifier avant
strcpy(buff,"x");	dépend de la valeur de *i* et la valeur retournée par malloc n'est pas vérifiée

Table 1: Type d'analyses faites par l'outil *TestInv*

Le tableau 1 représente les types d'analyse faits par l'outil sur une sélection de lignes de code d'un exemple de programme contenant la vulnérabilité considérée. Notons que si la taille de *i* est vérifiée avant l'appel à la fonction *malloc* et que la valeur retournée est également vérifiée, alors nous n'aurons aucune vulnérabilité. La vulnérabilité n'est signalée que si l'ensemble des conditions nécessaires sont vérifiées: *i* est teintée, la taille de *i* n'est pas vérifiée, et *buff* n'est pas vérifié avant utilisation.

La Figure 4 montre les traces de l'exemple et les diagnostics produits par l'outil. Les endroits où l'outil détecte les variables teintés sont signalés par des commentaires en italique. Comme TestInv vérifie toutes les conditions avant de signaler la présence d'une vulnérabilité, le nombre de faux positifs est nettement réduit comparé à d'autres outils existants qui pourraient être utilisés pour détecter ce type de vulnérabilités comme Valgrind avec le plug-ins memcheck [NS07] (qui détecte si un buffer est mal utilisé) et flayer[§] (qui détecte si une variable est taintée)

[‡] Cette vérification est utile pour la condition modélisant le nœud 3 qui appartient à d'autres scénarios non détaillés dans ce papier.

[§] http://www.code.google.com/p/flayer

```
...
bytes_read = read(fd, tmp, 6);
 80483e5:     mov   DWORD PTR [esp+0x8],0x6
 80483ed:     lea   eax,[ebp-0x1b]                <-- va être teintée par l'outil car la valeur
                                                       depend d'un fichier extern
 80483f0:     mov   DWORD PTR [esp+0x4],eax
 80483f4:     mov   eax,DWORD PTR [ebp-0x14]
 80483f7:     mov   DWORD PTR [esp],eax
 80483fa:     call  804831c <read@plt>
 80483ff:     mov   DWORD PTR [ebp-0x10],eax  <-- va être teintée par l'outil car la valeur de
                                                       retour depend d'un fichier extern
/home/ed/test.c:12
tmp[bytes_read]='\0';
 8048402:     mov   eax,DWORD PTR [ebp-0x10]
 8048405:     mov   BYTE PTR [ebp+eax-0x1b],0x0  <-- utilisation d'une variable teintée
                                                       comme position pour changer un buffer
FOUND VULNERABILITY: use_of_tainted_value_to_change_buffer : line /home/ed/test.c:12
/home/ed/test.c:13
int i=atoi(tmp);
 804840a:     lea   eax,[ebp-0x1b]
 804840d:     mov   DWORD PTR [esp],eax
 8048410:     call  804832c <atoi@plt>
 8048415:     mov   DWORD PTR [ebp-0xc],eax   <-- va être teintée par l'outil car la valeur
                                                       depend d'une variable teinté
/home/ed/test.c:15
char *buff;
buff=malloc(i);
 8048418:     mov   eax,DWORD PTR [ebp-0xc]
 804841b:     mov   DWORD PTR [esp],eax
 804841e:     call  804833c <malloc@plt>
 8048423:     mov   DWORD PTR [ebp-0x8],eax
FOUND VULNERABILITY: use_of_tainted_value_to_determine_buffer_size : line /home/ed/test.c:15
/home/ed/test.c:16
strcpy(buff,"x");
 8048426:     mov   eax,DWORD PTR [ebp-0x8]
 8048429:     mov   WORD PTR [eax],0x78
FOUND VULNERABILITY: did_not_check_return_value : line /home/ed/test.c:16
...
Program terminated normally (Exit status: 0x0008)
Error disassembling next instruction (address: 0xB7DC74C0)
TOTAL NUMBER OF 3 VULNERABILITIES FOUND
```

Figure 4: Exemple de traces et des vulnérabilités détectées par *TestInv*

5 Travaux similaires

5.1 Analyse des causes racines

Notre méthode de modélisation des vulnérabilités est comparable à l'analyse des causes racines (Root Cause Analysis en anglais) des brèches de sécurité d'un logiciel en partageant plusieurs caractéristiques. Néanmoins, notre méthode se différencie sur les deux principaux points suivants. Notre méthode de construction de VCG ne s'intéresse pas uniquement au problème "*Qui a causé la vulnérabilité* ?" mais elle a pour objectif également de répondre à la question "*Qui aurait pu causer la vulnérabilité* ?". En effet, l'occurrence d'une vulnérabilité sur une application donnée ne permet pas toujours de déterminer l'ensemble des causes, une réflexion plus fine devient donc nécessaire afin de définir les différents scénarios qui peuvent causer la vulnérabilité considérée. De plus, notre méthode nécessite un haut degré de formalisation des causes pour que la détection automatique des vulnérabilités associées soit possible. Bien qu'utilisés dans certaines méthodes d'analyse des causes racines, les modèles formels ne constituent pas une exigence de ce type de méthodes.

5.2 Analyse des vulnérabilités

L'analyse de vulnérabilité ne se restreint pas à la modélisation de la vulnérabilité. Schumacher décrit les lignes directives d'un modèle général de cycle de vie d'un logiciel de sécurité [SAS00]. Néanmoins, le manque de détails nécessaires rend difficile sa mise en pratique. Nous pensons que la modélisation des vulnérabilités permet de pallier aux insuffisances des modèles de haut niveau comme SIFR.

La classification des vulnérabilités logicielles est un domaine de recherche très actif [AIS96, Bis99, Krs98]. La modélisation de vulnérabilités devrait exploiter ces efforts de classification puisqu'il s'avère que les vulnérabilités d'une même classe sont caractérisées par des causes assez similaires. Réciproquement, la classification des vulnérabilités peut tirer profit de la modélisation puisque les vulnérabilités à causes similaires appartiennent généralement à la même classe. Enfin, un nombre de travaux portant sur l'analyse de vulnérabilités spécifiques sont présentés dans [KST98, Spa89]. Les informations fournies par une telle analyse approfondie peuvent être utilisées dans les phases préliminaires de modélisation des vulnérabilités.

5.3 Techniques de test passif

La technique de test passif a été largement explorée par l'équipe de recherche de Télécom & Management SudParis pour la vérification de propriétés dans le domaine des protocoles. Dans [ACC$^+$04], un algorithme d'analyse de traces d'entrées/sorties d'un système vis-à-vis de sa spécification formelle FSM (Finite State Machines) est présenté. Cet algorithme est basé sur la technique de retour en arrière (backtracing) afin de reconstruire l'état passé du système à partir de son état courant. Cette technique a été également utilisée dans [BCNZ05] pour vérifier des propriétés attendues, exprimées en *invariants*, sur les traces de communication d'un protocole. De manière générale, un invariant désigne une expression régulière qui décrit une dépendance dans l'ordre d'apparition des événements. Un invariant permet d'exprimer, par exemple, qu'un événement donné e_1 doit être suivi (ou précédé) par un certain événement e_2. Enfin, l'approche de test passif a été aussi utilisée pour vérifier des règles de sécurité, exprimées en Nomad [CCBS05], sur des réseaux basés services [MBCB08]. Afin de montrer la faisabilité de cette approche, son application sur un cas d'étude réel, fourni par le groupe SAP[¶], a été réalisée.

Cet article étend et applique les idées développées au cours de nos travaux antérieurs sur le test passif pour la détection systématique de vulnérabilités en proposant une sémantique formelle des vulnérabilités et de leurs causes. La définition d'une telle sémantique formelle rend possible l'utilisation des outils de test passif pour l'analyse automatique de traces d'exécution d'un programme en vue de détection de vulnérabilités.

6 Conclusions et perspectives

L'utilisation des outils automatiques pour le test des aspects sécurité est fortement recommandée par les meilleures démarches de développement de produits logiciels. Dans cet article, nous proposons une spé-

¶ http://www.sap.com/

cialisation du concept de VCG afin de pouvoir détecter automatiquement, par les outils de test, les vulnérabilités qu'ils modélisent. Pour cela, nous définissons le langage formel VDC qui permet d'associer une sémantique formelle aux VCG et aux causes qu'ils incluent. Une telle modélisation formelle des VCG est utilisée par les outils de test pour une détection automatique de la présence des vulnérabilités considérées. Nous avons également montré comment on pouvait exploiter l'outil de test passif *TestInv* pour l'analyse de la trace d'exécution d'un programme en vue de détection de vulnérabilités potentielles. Notons que le taux de faux positifs est inversement proportionnel au degré de précision des VCG. Comme les modèles de vulnérabilités sont indépendants de l'outil de test, il devient possible à n'importe quel utilisateur de mettre à jour les modèles de vulnérabilités déjà existants en ajoutant de nouveaux modèles. Ceci est un avantage indéniable par rapport aux outils commerciaux existants où des modèles de vulnérabilités ne peuvent être ajoutés ou mis à jour que par leurs propriétaires.

Afin de montrer la faisabilité de notre approche, nous travaillons actuellement sur l'application de notre méthode pour la modélisation de différents types de vulnérabilités logicielles. Nous envisageons également d'apporter quelques améliorations à l'outil de test Passif *TestInv* afin qu'il soit capable de déduire encore plus d'informations sur les variables du code à analyser.

References

[ABS06] S. Ardi, D. Byers, and N. Shahmehri. Towards a structured unified process for software security. In *Procedeengs of the ICSE 2006 Workshop on Software Engineering for Secure Software (SESS06)*, Shanghai, China, 2006.

[ACC+04] B. Alcalde, A. R. Cavalli, D. Chen, Davy Khuu, and David Lee. Network protocol system passive testing for fault management: A backward checking approach. In *FORTE*, pages 150–166, 2004.

[AIS96] Taimur Aslam, Krsul Ivan, and Eugene Spafford. Use of a taxonomy of security faults. In *Proceedings of the 19th National Computer Security Conference*, 1996.

[BASD06] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling software vulnerabilities with vulnerability cause graphs. In *Procedeengs of the International Conference on Software Maintenance*, Philadelphia, PA, USA, 2006.

[BCNZ05] E. Bayse, A. Cavalli, M. Nńez, and F. Zaïdi. A passive testing approach based on invariants: application to the wap. *Computer Networks and ISDN Systems*, 48(2):247–266, 2005.

[Bis99] Matt Bishop. Vulnerabilities analysis. In *Web Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection*, 1999.

[BS07] D. Byers and N. Shahmehri. Design of a process for software security. In *Proceedings of the Second International Conference on Availability, Reliability and Security, ARES2007*, Vienna, Austria, 2007.

[CCBS05] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: A Security Model with Non Atomic Actions and Deadlines. In *Computer Security Foundations Workshop (CSFW)*, pages 186–196, 2005.

[CMML08] A. R. Cavalli, E. Montes de Oca, W. Mallouli, and M. Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *The 12th IEEE International Symposium on Distributed Simulation and RealTime Applications*, Vancouver, Canada, October 2008. IEEE Computer Society.

[Cov08] Coverity. Prevent, 2008. http://www.coverity.com/ (accessed September).

[For08] Fortify Software. Fortify SCA, 2008. http://www.fortifysoftware.com/products/sca (accessed September).

[Klo08] Klocwork. K7, 2008. http://www.klocwork.com(accessed September).

[KMC06] Chunguang Kuang, Qing Miao, and Hua Chen. Analysis of software vulnerability. In *ISP'06: Proceedings of the 5th WSEAS International Conference on Information Security and Privacy*, pages 218–223, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[Krs98] I. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998.

[KST98] I. Krsul, E. Spafford, and M. Tripunitra. An analysis of some software vulnerabilities. In *Proceesings of the 21st NIST-NCSC National Information Systems Symposium*, pages 111–125, 1998.

[LNS+97] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *ICNP '97: Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, page 113, Washington, DC, USA, 1997. IEEE Computer Society.

[MA01] R. E. Miller and K. A. Arisha. Fault identification in networks by passive testing. In *Advanced Simulation Technologies Conference (ASTC)*, pages 277–284. IEEE Computer Society, 2001.

[MBCB08] W. Mallouli, F. Bessayah, A. Cavalli, and A. Benameur. Security rules specification and analysis based on passive testing. In *The IEEE Global Communications Conference (GLOBECOM 2008)*, 2008.

[NS07] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[S. 04] S. Redwine and N. Davis. Processes to produce secure software. 2004. Task Force on Security Across the Software Development Lifecycle, Appendix A.

[SAS00] M. Schumacher, R. Ackermann, and R. Steinmetz. Towards security at all stages of a system's life cycle. In *Proceedings of the International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2000.

[Spa89] Eugene Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19(1), 1989.